C++/CLI – Why, oh why?

Seb Rose
Claysnow Limited
ACCU 2008

Roadmap

- Apology – less code, more words than promised

- Background
- Brief Syntax tour
- Interoperability tour
- Some small examples
- The rest

Introduction

- What is C++/CLI?
- Why does it exist?
- When should it be used?
- Who should use it?
- Will I regret it?

Background

- .NET is similar to a virtual machine
- Managed execution environment called Common Language Infrastructure (CLI)
- CLR is implementation of CLI
- JIT compilation of Common Intermediate Language (CIL) – formerly MSIL
- Assembly is unit of deployment
- Metadata describes contents of assembly

Common Type System

- Common Type System defines type system of the CLI
- Many languages target CLI (i.e. provide compilers that output assemblies)
- To facilitate interoperation between languages the Common Language Specification (CLS) was defined
- CLS is a subset of the CTS

.NET languages

- Popular .NET languages are C# and VB
- Much functionality is exposed by .NET Framework libraries
- Other Win32 functionality can be accessed using Platform Invoke (P/Invoke)
- P/Invoke requires .NET declaration of functions to be used.
- Types need to be marshalled

C++ Managed Extensions

- Shipped with Visual Studio .NET
- AKA Managed C++
- New keywords started with double underscores
- Attempted to elide differences between CTS and C++ type systems
- Proved very unpopular with developers

C++/CLI Rationale

- Herb Sutter's rationale available in full at: http://www.gotw.ca/publications/C++CLIRationale.pdf

1) Language support for special code generation
2) Hide unnecessary differences, but expose essential differences
3) Don't interfere with evolution of ISO C++
4) Keywords don't have to be reserved words

C++/CLI Standardisation

- Most of the .NET development has been standardised
- C++/CLI was standardised by ECMA (ECMA-372)
- Objections from many national bodies due to fast-tracking of potentially confusing, divergent standard

Why C++/CLI

- Easier Interop with native C++
  - "It Just Works" (IJW) design intent
- Most powerful .NET language (?)
  - we'll see some of the language constructs in the extensions to C++

- Not available for Compact Framework

Why Interop?

- Vast investment in existing software means we can’t just throw it away
- New functionality may only be available in managed environment
- Managed development promises enhanced productivity
- How can interop be made easier?

Easier Interop

- A lot of the legacy codebase was implemented in C++
- Access to this functionality used to be as easy as including a header file and linking against an export library
- P/Invoke declarations could be created for each library, but they expose methods, not types, and limited marshalling control.

C++/CLI Compatibility

Visual Studio complier/linker provides 4 build models:

1) Native – normal behaviour
2) CLR – compiles standard C++ and C++/CLI to CIL (and can link native object files too)
3) CLR:pure – compiles standard C++ and C++/CLI to CIL (no native object files)
4) CLR:safe – only compiles C++/CLI

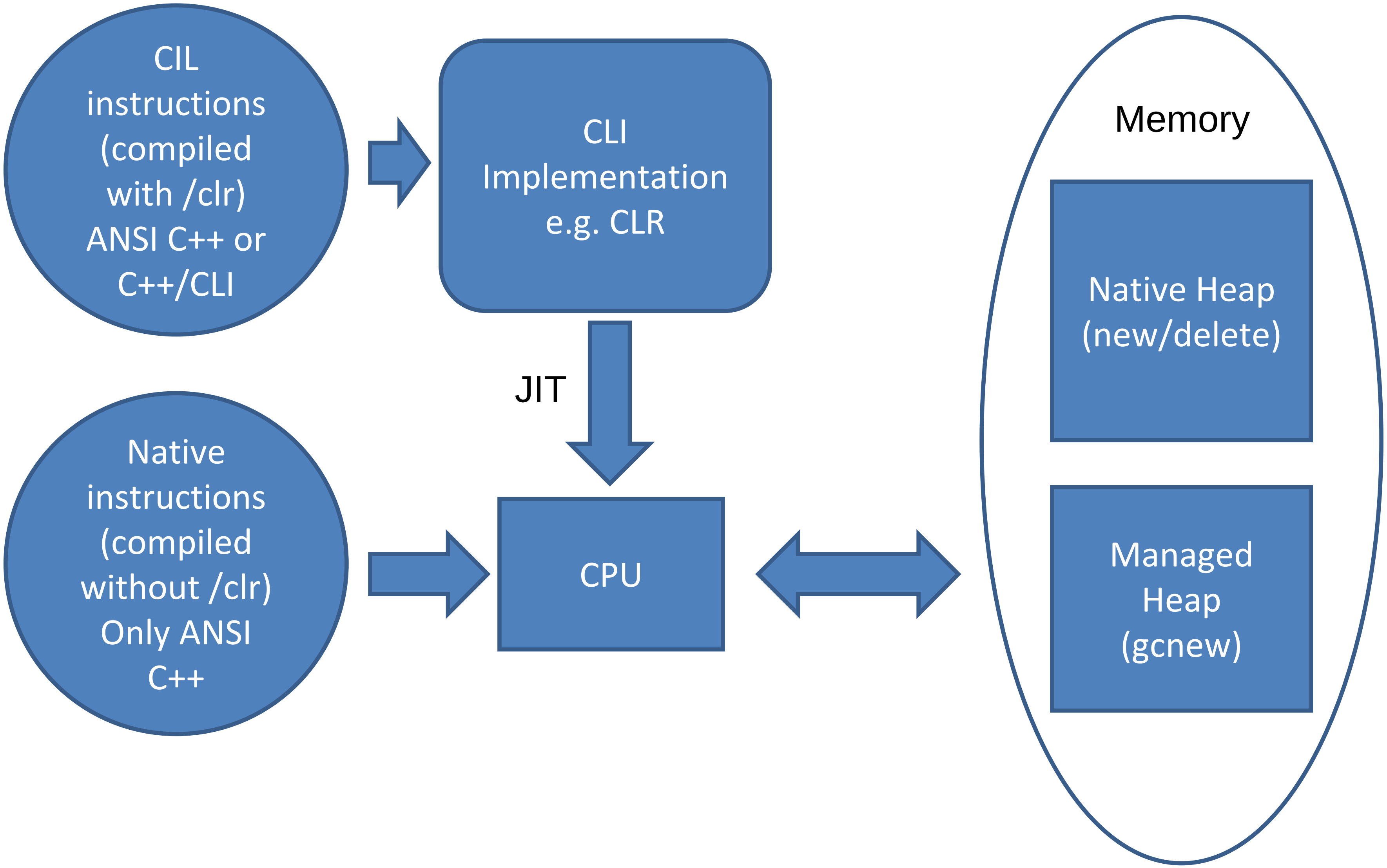
Compatibility Types

- /clr gives source file and object file compatibility
- /clr:pure gives source file compatibility
- /clr:safe gives no compatibility, but enables the use of C++/CLI as a first class .NET language with verifiability etc.

clr:pure

- Native calling conventions not allowed, so not callable from native code
- What it’s for:
  - mixed code assemblies must be stored in files
  - mixed code EXEs cannot be loaded dynamically into a process

Schematic



CIL instructions (compiled with /clr) ANSI C++ or C++/CLI
CLI Implementation e.g. CLR
JIT
Native instructions (compiled without /clr) Only ANSI C++
CPU
Memory
Native Heap (new/delete)
Managed Heap (gcnew)

Again, in words

<u>Managed Type != Managed Code</u>

- Managed Types are always garbage collected
- Native Types are never garbage collected
- Methods for Managed Types are always compiled to CIL
- Methods for Native Types may be compiled to CIL or native opcodes

Notes for .NET developers

- C++ is very different from C#. C++/CLI is very different from C++. Steep learning curve.
- Visual Studio Intellisense not nearly as clever
- Code marked with Conditional(“Debug”) attribute is included in C++/CLI release builds.
- A C++/CLI destructor is not a .NET finalizer. A finalizer can be defined as: Foo::!Foo() {}

New Syntax – Type System

- All types inherit from System::Object
- Primitives are automatically boxed when used in reference contexts
- **value** defines a value type that inherits from System::ValueType
- Also **interface** and **enum**
- New visibilities: **internal**, **public protected**, **protected private**

New Syntax – Type System

- Single inheritance
- May implement any number of **interface**s
- Managed class definition:
  public ref class Foo {};
- Tracking handle: Foo**^** foo = **gcnew** Foo();
- Tracking reference:
  void createFoo(Foo^**%** foo) {
      foo = gcnew Foo();
  }

New Syntax – Object Creation

- If you call a **virtual** method during construction of a C++/CLI class, it will call the most derived method, even though the most-derived constructor has not yet been called.
- In C++/CLI, member field initialisation takes place before calling any base class constructor.
- To avoid problems, prefer member initialisation over explicit initialisation in the constructor.

New Syntax – Object Destruction

- The runtime manages memory, but the developer still manages resources.
- The .NET idiom for resource release is to implement IDisposable::Dispose()
- The compiler will map a C++/CLI destructor to Dispose()
- The compiler maps a call to **delete** a C++/CLI instance to a call to Dispose()

New Syntax – Object Destruction

- Managed destructors may be called multiple times
- All calls after first must be ignored. Consider whether class needs to be thread safe.
- Calls to other methods on objects that have been disposed can throw an **ObjectDisposedException**
- Use GC::KeepAlive to prevent finalization

New Syntax – Implicit Dereference

- C++/CLI allows you to use RAII (Resource Acquisition Is Initialisation)
- Compiler translates:

```
void doSomething(int i)
{
   Foo foo(i);

   foo.bar();
}
```

```
void doSomething(int i)
{
   Foo^ foo = gcnew Foo(i);

   try {
     foo->bar();
   }
   finally {
      delete foo;
   }
}
```

New Syntax - Dispose pattern

```
ref class Foo
{
public:
  ~Foo() {}
  !Foo() {}
};
```

```
ref class Foo : Idisposable
{
 public:
   virtual void Dispose() sealed {
      Dispose (true);
      GC::SuppressFinalize(this);
   }
 protected:
    virtual void Finalize() override {
      Dispose(false);
    }
    virtual void Dispose(bool Disposing) {
       if (disposing)
          ~Foo();
       else
          !Foo();
    }
 private:
      // User supplied destructor & finalizer
};
```

New Syntax - properties

```
property bool IsHappy {
  bool get() { return isHappy_;}
  void set(bool isHappy) { isHappy_ = isHappy; }
}
```

EQUIVALENT TO:

```
property bool IsHappy;

this->IsHappy = true;
```

New Syntax - Modifiers

- **abstract**
  - can be applied to classes and methods
  - similar to pure virtual (=0), but may not have an implementation
  - must be applied to classes with abstract method(s)
- **sealed**
  - can be applied to classes and methods
  - prevents further derivation/overriding

New Syntax – More Modifiers

- **virtual -** introduces a virtual method:
  ```
  virtual void f();
  ```
- **override -** overrides a virtual method:
  ```
  virtual void f() override;
  ```
- **new** – introduces new virtual ‘slot’
  ```
  virtual void f() new;
  ```
- Named overriding:
  ```
  virtual void another_f() = Base::f;
  ```

New Syntax - const

- Say goodbye to **const**.
- You cannot declare methods as **const**.
- You can declare parameters as **const**, but without **const** methods you cannot call any methods on the object.
- You can declare fields as **const**, u this is rarely useful – use **initonly** or **literal**.
- **const** only makes sense for local primitives

Arrays and auto_handle

- msclr::auto_handle
  – analagous to std::auto_ptr
- cli - pseudo namespace
  - array<int>^ my;
  - my = gcnew array<int,1>(2);
  - interior_ptr<int> pi = &(my[0]);

Mixing the type systems

- Managed classes cannot contain native members, but can contain pointers
- Native classes cannot contain managed members but you can use **msclr::gcroot<>** and **msclr::auto_gcroot<>**
- Use **cli::pin_ptr<>** to obtain a pointer to a managed object
- Can manually create auto pointer for native to manage reference to managed object

SafeHandle

- Utility base class that manages native resources reliably in the presence of Asynchronous exceptions
- Uses Constrained Execution Regions (CER) to guarantee successful allocation
- Protects against "handle-recycling" exploit:
  http://blogs.msdn.com/bclteam/archive/2006/06/23/644343.aspx

Marshalling

- System::Runtime::InteropServices::Marshal provides many methods for marshalling
- Some require matching calls to relevant Marshall::FreeXxxx methods
- Visual Studio 2008 ships with a simpler marshall_as<> template library that can be specialised for user types.
- Marshalling contexts provide scoped resource management

SEH Exceptions

- Can perform SEH __try handling in managed code
- Automatic translation via **_set_se_translator** doesn’t happen in managed code
- Automatic translation to **SEHException** or one of the specific derived exceptions (e.g. **OutOfMemoryException**)

C++ & C++/CLI Exceptions

- Can mix in a single **try** block can have **catch** blocks for managed and native exceptions
- Catch native exceptions before managed exceptions or they may be translated into **SEHException**
- You can catch a managed exception in native code using an SEH **__try** statement, but you will not get access to its data

Templates

- Templates are usually defined in header files
- Template members depending on compilation model of file including template
- You can easily end up with native and managed instantiations of same template
- Linker chooses the one that matches compilation model of caller

Converting a C++ project

- Must use DLL versions of CRT
- Apply /clr at file level
- Need separate PCH file for managed files
- /EHs compiler switch (no SEH) not allowed – change to EHa at project level
- /ZI compiler switch (Edit & Continue) not allowed – change to Zi at project level

Converting a C++ Project 2

- CLR required (not supported by Mono?)
- Requires CLR 2.0 or later
- Only one version of CLR can be loaded into a process – can specify **requiredRuntime** in configuration file
- *RegisterOutput:false* for linker – cannot load mixed EXEs dynamically
- Default COM apartment initialisation often wrong

CAS Policies

- Code Access Security - .NET safety feature
- Default security policy loads applications from network drives in a sandbox with restricted permissions
- Mixed or pure assemblies are not verifiable, so cannot load in sandbox
- Could use caspol.exe to grant assembly rights, except that it uses reflection, and mixed EXEs cannot be loaded dynamically

Function Interop

- Any combination of call can be made
- Thunks automatically perform transition
- Native->Managed thunks are created automatically at assembly load time
- Managed->Native thunks are created dynamically on demand by JIT compiler

Native->Managed Thunks

- .vtfixup in assembly metadata for each method with native calling convention
- Interoperability vtable in assembly that maps each method to a native->managed thunk
- At load time CLR creates a thunk for each .vtfixup and stores pointer to it in vtable
- Thunk only used when caller is native

Native->Managed Thunks 2

- Not generated for methods with _clrcall calling convention

1) All members of Managed types are _clrcall

2) Instance members of Unmanaged types _clrcall or _thiscall depending on args

3) Static/global methods _clrcall or _cdecl depending on args

4)_stdcall allowed in 2) and 3) above

Native->Managed Thunks 3

- Calling a C++ class compiled using /clr from native code required a transition
- C++ class methods are exposed as mangled global functions with a **this** pointer
- Function pointers to managed code (with native calling convention) will be pointers to thunks
- Similarly, pointers to thunks are in the vtable of C++ classes compiled to managed code

Double Thunking

- Function pointers and vtables to C++ methods compiled to managed code point to thunks
- If called by managed code there needs to be a managed->native thunk before the native->managed thunk can be called: double thunk!
- Function pointers can be cast to _clrcall
- Virtual functions can be declared with _clrcall, but this must be done when function introduced (and closes door to native callers)

Managed->Native Thunks

- P/Invoke metadata generated automatically
- Type compatibility means reduced marshalling
- Three possible thunk types:
  1) Inlined thunks – saves cost of function call
  2) Non-inlined thunks
  3) Generic thunks – special marshalling available, though only by using custom metadata

Managed->Native Thunks 2

- If native function is in a DLL the generated thunk will assume that it might use SetLastError
- Thunk will never be inlined
- Result of GetLastError stored in TLS
- Could use linker /CLRSUPPORTLASTERROR:NO
- Better to define custom metadata: [DllImport(..., SetLastError=false)]void func();

GetLastError gotchas

- If local native methods use SetLastError, then error will be lost, because P/Invoke doesn’t store error code in TLS
- If native function from DLL is called through a function pointer, then thunk will be inlined and error might be lost, because P/Invoke doesn’t store error code in TLS

Delegates and function pointers

- Marshall::GetFunctionPointerForDelegate converts managed handler to a callback that can be passed to a native API
- Call **ToPointer()** to get function pointer
- You must ensure that the delegate doesn’t get garbage collected while the callback is in use
- GetDelegateForFunctionPointer allows native code to be called as-if it were a delegate

Application Startup

- OS looks for PE entry point
- Native apps typically use mainCRTStartup (or similar) from msvcrt.lib
- CLR apps use _CorExeMain from mscoree.lib, which:
  - loads & starts CLR
  - initialises the assembly and executes the Module Constructor
  - calls the entry point of the assembly

Module Constructor

- Signature: void _clrcall .cctor()
- Can be manually provided if CRT not required
- Default implementation initialises the CRT:
  - initialises vtables
  - parses command line
  - global data in native code is initialised
  - global data in managed code is initialised
- Note: changing the compilation model of a file can change order of global data initialisation

DLL Startup

- Mixed code DLL entry point is _CorDllMain which then calls _DllMainCRTStartup
- DLL entry point can be called whenever a DLL is loaded or unloaded or a thread is started/shutdown
- DllMain is then called
- _CorDllMain fixes up the interoperability vtable to delay load the CLR if a managed function is called and the CLR isn't loaded yet

DllMain and the Loader Lock

- The OS acquires the loader lock before calling _CorDllMain
- User implementations of DllMain must not:
  - do inter-thread communication
  - attempt to load another library explicitly
  - execute managed code
- Also, since _DllMainCRTStartup initialises global variables, their ctors and dtors should observe the same restrictions

Dll Module Constructor

- Module Constructor is called after the loader lock has been released
- If a source file is compiled with /clr all global objects are initialised by the Module Constructor
- Caution: If a global defined in a /clr file is accessed by native code, then it may not yet be initialised, because the CLR may not have been delay loaded

Wrapping a Native DLL

- It normally doesn’t make sense to expose the native API ‘as is’
- Expose .NET idioms not Win32 (or others)
  - properties
  - events
  - exceptions
- Create a mixed MFC Regular DLL to wrap a MFC Extension DLL

CLS Type Compliance

CLSCompliantAttribute:

- Names not distinguished by case
- No global static fields or methods
- Exceptions derived from System::Exception
- No unmanaged pointer types
- No boxed value types
- Custom attributes only of types Bool, Char, String, Int, Single, Double, Type

Calling COM Objects - RCW

- Create Runtime Callable Wrapper using tlbimp.exe
- Dependency on RCW assembly/DLL
- Signatures are direct conversions of COM functions

Custom RCW

- Fuller control of managed interface
- Store reference to COM object in **msclr::com::ptr** instance
- Provide custom API and marshalling
- HRESULTS can be converted to exceptions using **Marshall::GetExceptionForHR**

Calls from COM Objects - CCW

- Assembly needs to be registered (regasm.exe)
- **#import** the type library (.tlb)
- **AddRef**, **Release**, **QueryInterface** called automatically
- Classes must have default constructor
- Return values translated to **out** references
- Runtime handles marshalling, but need to release native resources

WinForms/MFC Interop

- afxwinforms.h contains utility classes to allow use of WinForms in MFC:
  - CWinFormsControl
  - CWinFormsView
  - CWinFormsDialog
- Can create a WinForms User Control that allows use of MFC controls on WinForms
- You can also interop WPF with MFC

Events and delegates

- Event handlers cannot be native member functions (can be global/static functions)
- Use MAKE_DELEGATE(HandlerType, handler);
- BEGIN_DELEGATE_MAP(class_name)
  EVENT_DELEGATE_ENTRY(handler, Object^, HandlerArgs^)
  END_DELEGATE_MAP()

Not using CRT?

- Compile with /Zl (Omit Default Library Names)
- Implement your own Module Constructor:
  #pragma warning(disable:4483)
  void _clrcall _identifier(".cctor")() {}
- Ensure that _CorExeMain is resolved:
  #pragma comment(lib, "mscoree.lib")
- Specify your own managed entry point:
  #pragma comment(linker, "/ENTRY:MyEntry")
- Remember not to use any CRT methods!

Single binary – multi language

- Can create a single assembly application from source code written in C#, managed C++/CLI and native C++
- Cannot be built from Visual Studio
- Requires use of *netmodules* and command line compilation/linking

[Teixeira]

Single DLL for Native and Managed

- Mixed mode DLL (/clr)
- Conditional __MANAGED__ compilation in header of **gcroot<>** or **intptr_t**
- Public API must only use native types
- Managed API includes operator to access underlying managed type

Managed Types and Static Libraries

- Identity of managed types is dependant on assembly they are defined in
- Linker seems unable to resolve reference: LNK2020
- Microsoft says this is side effect of IJW
- If C++ type defined in same source file & instantiated by caller, then linker resolves reference. Go figure.

[Sanna]

Summary

- ‘Safe’ C++/CLI gives you much of the power of C++ in a Windows .NET environment (e.g. Templates and deterministic resource management)
- C++/CLI gives you a lot of options to interop with native/legacy code at the price of added complexity

References

[Heege 2007]	Expert C++/CLI	Apress	1-59059-756-7
[Sivakumar 2007]	C++/CLI in Action	Manning	1-932394-82-8
[Duffy 2006]	Professional .NET 2.0	Wrox	978-0-7645-7135-0

[Teixeira 2007] Linking native C++ into C# Applications
http://blogs.msdn.com/texblog/archive/2007/04/05/linking-native-c-into-c-applications.aspx

[Zhang Blog] Netmodule vs. Assembly
http://blogs.msdn.com/junfeng/archive/2005/02/12/371683.aspx

[MSDN] .netmodule Files as Linker Inputs
http://msdn2.microsoft.com/en-us/library/k669k83h.aspx

[Heege Blog] Marshalling native function pointers
http://www.heege.net/blog/PermaLink,guid,94167d73-7954-4a5c-a745-dc60d352cdef.aspx

[Sanna Forum] Managed Classes in Static libs
http://forums.microsoft.com/MSDN/ShowPost.aspx?PostID=463461&SiteID=1